Automated Program Repair
Using Formal Verification Techniques

Orna Grumberg
Technion, Israel

CREST Open Workshop (COW), London
January 20, 2020
Model Checking

• Given a system and a specification, does the system satisfy the specification.
Automated program repair

• **Model checking** finds bugs in the program
  • **Bug**: A program run that violates the specification

• **Repair tool** automatically suggests repair(s)
  • **Repair**: Changes to the program code, resulting in a correct program
In this talk – two approaches

• Exploit Model Checking technologies for program repair
  • Mutation-Based Program Repair
  • Assume, Guarantee or Repair
Sound and Complete Mutation-Based Program Repair

[Rothenberg, Grumberg, FM’16+work in progress]
Mutation-Based Program Repair

Sequential program

Assertions in code

Given set of mutations

Can we use these mutations to make all assertions hold?

Assignments, conditionals, loops and function calls

Assertion violation

operator replacement (+ → -), constant manipulation (c → c + 1)

Return all possible repairs
Example

```c
int f(int x, int y) {
    int z;
    if (x + y > 8) {
        z = x + y;
    } else {
        z = 9;
    }
    if (z >= 9) z = z - 1;
    assert(z > 8);
    return z;
}
```

$x = 5, y = 2$

1. $int z;$
2. $if (x + y > 8) { \text{ }$
3. $z = x + y;$
4. $} \text{ } else \{ \text{ }$
5. $z = 9;$
6. $\}$
7. $if (z \geq 9) z = z - 1;$
8. $assert(z > 8);$
9. $return z;$

$x = 9$

$z = 9$

$z = 8$
Example

```c
int f(int x, int y){
    int z;
    if (x + y > 8) {
        z = x + y;
    } else {
        z = 9;
    }
    if (z ≥ 9) z = z + 1;
    assert(z > 8);
    return z;
}
```

Mutation list:
- Replace + with –
- Replace – with +
- Replace > with ≥
- Replace ≥ with >

Repair list:
- **option 1:**
  - line 7: replace ≥ with >
- **option 2:**
  - line 7: replace – with +

Note: Repairs are minimal
Example

```c
int f(int x, int y){
1.    int z;
2.    if (x + y > 9) {
3.        z = x + y;
4.    } else {
5.        z = 10;
6.    }
7.    if (z ≥ 9) z = z – 1;
8.    assert(z > 8);
9.    return z;
}
```

Mutation list:
- Replace + with –
- Replace – with +
- Replace > with ≥
- Replace ≥ with >

Increase constants by 1.

At this point z ≥ 10
Overview of our approach

Finding all correct programs from a finite set of programs
Finding all unsatisfiable constraint sets from a finite set of constraint sets

Input:
a buggy program

Output:
All minimal repairs, sorted by size

```
int f(int x, int y){
    int z;
    if (x + y > 8) {
        z = x + y;
    } else {
        z = 9;
    }
    if (z >= 9) z = z - 1;
    assert(z > 8);
    return z;
}
```
First step - Translation

Goal: Translate the program into a set of constraints which is satisfiable iff the program has a bug (i.e. there exists an input for which an assertion fails)

Work by Clarke, Kroening, Lerda (TACAS 2004) (CBMC)
- Simplification
- Unwinding of loops
  - a bounded number of unwinding
- Conversion to SSA
First step - Translation

```c
int f(int x, int y){
    int z;
    if (x + y > 8) {
        z = x + y;
    } else {
        z = 9;
    }
    if (z ≥ 9) z = z - 1;
    assert(z > 8);
    return z;
}
```
First step - Translation

```c
int f(int x, int y){
    int z;
    if (x + y > 8) {
        z = x + y;
    } else {
        z = 9;
    }
    if (z ≥ 9) z = z - 1;
    assert(z > 8);
    return z;
}
```

```c
{ g₁ = x₁ + y₁ > 8,
  z₂ = x₁ + y₁,
  z₃ = 9,
  z₄ = g₁?z₂:z₃,
  b₁ = z₄ ≥ 9,
  z₅ = z₄ - 1, 
  z₆ = b₁?z₅:z₄, 
  z₆ ≤ 8
}
```
Second step - Mutation

```c
int f(int x, int y){
    int z;
    if (x + y > 8) {
        z = x - y;
    } else {
        z = 9;
    }
    if (z ≥ 9) {
        z = z - 1;
    }
    assert(z > 8);
    return z;
}
```

Mutation list:
- Replace + with –
- Replace – with +
- Replace ≥ with >
Third step - Repair

\[ z_{0} = z_{0} - 1 \]
\[ z_{1} = z_{1} + 1 \]
\[ b_{0} = z_{1} > 9 \]
\[ z_{2} = x_{1} + y_{1}, z_{2} = x_{1} - y_{1} \]
\[ z_{3} = 9 \]
\[ z_{4} = g_{1} \? z_{2} : z_{3} \]
\[ b_{1} = z_{4} \geq 9, b_{1} = z_{4} > 9 \]
\[ z_{5} = z_{4} - 1, z_{5} = z_{4} + 1 \]
\[ z_{6} = b_{1} \? z_{5} : z_{4} \]
\[ z_{6} \leq 8 \]
Making repair more efficient

Goal: reducing the search space

1. When a **correct mutated program** is generated
   (Validate succeeds)
   • Eliminate non-minimal correct mutated programs

2. When a **buggy mutated program** is generated
   (Validate fails)
   • Eliminate “similar” buggy mutated programs
Buggy mutated program

Unsuccessful repair:
A set of mutations $M$ that results in a buggy program

Elimination:
• Find a small explanation $S$ for the bug
  • $S$ is a set of statements in the code
• Disallow any mutated program, containing $S$
Fault localization

Fault localization: A (small) explanation $S$ to a bug

In most works:
- *May* explanation
  - Changes to statements from $S$ *may* result in a repaired program
Fault localization

Fault localization: A (small) explanation $S$ to a bug

In our work:

• **Must** explanation
  • If *none* of the statements in $S$ is changed, then
    • regardless of changes applied to other statement
    • the same bug will remain
  • $\Rightarrow S$ must be changed
Fault localization: example

```c
int f(int x, int y){
  1. int z;
  2. z = x
  3. if (x >= 0) {
  4.       x = x + 1; y = x + 2;
  5.   } else {
  6.       z = 9;
  7.   }
  7. assert(z > 0);
  8. return z;
}
```
Fault localization: example

```c
int f(int x, int y) {
    int z; int t;
    z = x
    if (x >= 0) {
        x = x + 1; y = x + 2;
    } else {
        z = 9;
    }
    assert(z > 0);
    return z;
}
```

erroneous run:
- x=0, y=0
- z=0
- x=1, y=2
- z=0

Repair: line 3 should change to `(x > 0)`
**Fault localization by slicing**

```c
int f(int x, int y){
    int z; int t;
    z = x
    if (x >= 0) {
        x = x + 1; y = 0;
    } else {
        z = 9;
    }
    assert(z > 0);
    return z;
}
```
**Theorem:** Our algorithm is sound and complete

That is, no good repair is eliminated by our search space pruning
Summary

*Mutation-based formal repair is not as scalable as test-based approaches*

- It can assist a programmer in debugging in initial stages of development
  - When bugs are simple, but many

- It also can help beginner programmers
  - Educational tool for students (experiments on Codeflaw)

- **Pruning** has a significant impact in making the search for repair(s) efficient
Assume, Guarantee or Repair

[Frenkel, Grumberg, Pasareanu, Sheinvald, TACAS’2020]
Goal

• Exploit the partition of the system into components

• Compositional model checking verifies small components and conclude the correctness of the full system

• If a vulnerability is found, repair is applied to one of the components
Communicating systems

- C-like programs
- Described as a control-flow graph (automaton)
- Use automata learning algorithms

```plaintext
1: while (true)
2:    pass = readInput;
3:    while (pass ≤ 999)
4:       pass = readInput;
5:    pass2 = encrypt(pass);
6:    return pass2;
```
Example

- Components synchronize over common channels
Example

- **Components synchronize over common channels**

\[ M_1 \]

\[ M_2 \]
Example

• Components synchronize over common channels

\[ M_1 \]

\[ p_0 \xrightarrow{enc(pass')} p_1 \xrightarrow{pass' = (pass') \cdot 2} p_2 \xrightarrow{return(pass')} \]

\[ M_2 \]

\[ q_0 \xrightarrow{read(pass)} q_1 \xrightarrow{pass \leq 999} \] (999 < pass) \[ q_2 \xrightarrow{read(pass)} \]

\[ q_0 \xrightarrow{return(pass2)} q_4 \xrightarrow{enc(pass)} \]
Example

- Components synchronize over common channels
Example

- Components synchronize over common channels
Example

• Components synchronize over common channels
Specifications

- Safety requirements – given as an automaton
- Behavior of the program through time
- “the entered password is different from the encrypted password”
- “there is no overflow”
Assume-Guarantee (AG) Rule

1. check if a component $M_1$ guarantees $P$ when it is a part of a system satisfying assumption $A$.
Assume-Guarantee (AG) Rule

1. check if a component $M_1$ guarantees $P$ when it is a part of a system satisfying assumption $A$
2. show that the other component $M_2$ (the environment) satisfies $A$
Assume Guarantee or Repair

counterexample - strengthen assumption

Learning

Model Checking

1. \( A_i \parallel M_1 \models P \)
   - true
   - false

2. \( M_2 \models A_i \)
   - true
   - false
   - \( \text{cex} \notin L(A_i) \)

real error?

Y

P violated in \( M_1 \parallel M_2 \)

Repair \( M_2 \)

N

P holds in \( M_1 \parallel M_2 \)

cex \( \parallel M_1 \neq P \)?
Semantic Repair

• The counterexample contains constraint -- violating overflow
• learn a constraint $C$ such that:
  • $C \land \text{pass} > 999 \land \text{pass2} = \text{pass} \cdot 2 \land \text{pass2} \geq 2^{64} \rightarrow \text{false}$
  • $C$ is over the input variables - pass.

$C := \forall \text{pass2}: \text{pass} > 999 \land \text{pass2} = \text{pass} \cdot 2 \land \text{pass2} \geq 2^{64} \rightarrow \text{false}$

• After quantifier elimination & simplification: $C = \text{pass} < 2^{63}$

Abduction - "Logical Magic"
1: while (true)
2:     pass = readInput;
3:     while (pass ≤ 999 or pass ≥ 2^{63})
4:         pass = readInput;
5:     pass2 = encrypt(pass);
6:     return pass2;
Syntactic repair

• The counterexample $t$ contains no constraint
  • It consists of communication actions and assignments
• Abduction will not help

3 methods to removing counterexample $t$:
• Exact: remove exactly $t$ from $M_2$
• Approximate:
• Aggressive:
**Example - Syntactic Repair**

No self loop, cannot *read* more than once each time!

Multiple reads are allowed
Agressive Repair

- Remove accepting states (can make the language of $M_2$ empty)
Approximate Repair

- Add an intermediate state to eliminate bad traces
Exact Repair

- Remove bad traces one by one
- First bad trace spotted is \textit{read(pass), read(pass)}
Comparing Repair Methods (logarithmic scale)

#15, #16, #18, #19 apply also abduction
Summary

- Learning-based Assume Guarantee algorithm for infinite-state communicating programs
- Semantic and syntactic repair
- Experiments provide proof of concept
Summary

• Two approaches to automatic program repair
  • based on formal method technologies
Thank you